IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-6, Issue-7, July 2018 E-ISSN: 2347-2693

Effective Road Networks Using Clue Based Route Search

Shaik Sharmila[1*], U. Mohan Srinivas[2]

[1]Computer Science and engineering, Qis College of Engineering and Technology, Ongole
[2]Computer Science and Engineering, Qis College of Engineering and Technology, Ongole

[*]*Corresponding Author: shakeelasharru786@gmail.com*

Available online at: www.ijcseonline.org

Accepted: 06/Jul/2018, Published: 31/July/2018

Abstract— With the advances in geo-positioning technologies and location-based services, it is nowadays quite common for road networks to have textual contents on the vertices. Previous work on identifying an optimal route that covers a sequence of query keywords has been studied in recent years. However, in many practical scenarios, an optimal route might not always be desirable. For example, a personalized route query is issued by providing some clues that describe the spatial context between Pose along the route, where the result can be far from the optimal one. Therefore, in this paper, we investigate the problem of clue-based route search (CRS), which allows a user to provide clues on keywords and spatial relationships. First, we propose a greedy algorithm and a dynamic programming algorithm as baselines. To improve efficiency, we develop a branch-and-bound algorithm that prunes unnecessary vertices in query processing. In order to quickly locate candidate, we propose an AB-tree that stores both the distance and keyword information in tree structure. To further reduce the index size, we construct a PB-tree by utilizing the virtue of 2-hop label index to pinpoint the candidate. Extensive experiments are conducted and verify the superiority of our algorithms and index structures.

Keywords—Spatial keyword queries, clue, Point-of-Interest, travel route search, query processing

I. INTRODUCTION

With the rapid development of location-based services and geo positioning technologies, there is a clear trend that an increasing amount of geo-textual objects are available in many applications. For example, the location information as well as concise textual descriptions of some businesses (e.g., restaurants, hotels) can be easily found in online local search services (e.g., yellow pages). To provide better user experience, various keyword related spatial query models and techniques have emerged such that the geotextual objects can be efficiently retrieved. It is common to search a Point-of-Interest (PoI) by providing exact address or distinguishable keyword (i.e., only few PoIs contain the keyword) in a region which can uniquely pinpoint the location. For example, we type the address "73 Mary St, Brisbane" or the name "Kadoya" on Google Maps to find a Japanese restaurant in the CBD area. Some existing work [8], [15], [26], [31], [33], [35] extends such query to more sophisticated settings, such as retrieving a group of geo-textual objects (usually more than 2) or a trajectory covering multiple keywords. However, it is not uncommon that a user aims to find a PoI with less distinguishable keyword such as "restaurant", but she can only provide more or less spatio-textual context information around the PoI. Liu et al. [25] formalize such context information as clues and use them to identify the most promising PoIs. Different with their work, we aim to find a feasible route on road networks by using clues. Particularly, in this paper, we investigate a novel query type, namely clue-based route search (CRS), which allows a user to provide clues on textual and spatial context along the route such that a best matching route w.r.t. the clues is returned. More specifically, a CRS query is defined over a road network G, and the input of the query consists of a source vertex vq and a sequence of clues, where each clue contains a query keyword and a user expected network distance. A vertex contains a clue keyword is considered as a match vertex. The query returns a path P in G starting at vq, such that (i.) P passes through a sequence of match vertices (PoIs) w.r.t. the clues and (ii.) the network distances between two contagious matched vertices are close to the corresponding user specified distance such that the user's search intention is satisfied.

1.1 Application Scenarios

The existing solutions (e.g., [6], [22], [28]) for trip planning or route search problem are dealing with the scenarios when a user wants to visit a sequence of PoIs, each of which contains a user specified keyword. Different optimization constraints are proposed, and the goal is to find an optimal route with minimum cost. In general, the

cost can be of various different types, such as travel distance, time or budget. However, to the best of our knowledge, none of the existing solutions (e.g., [6], [22], [28]) on trip planning or route search can be applicable for solving CRS queries since the optimization needs to be conducted based on the clues. As an extension of traditional route search queries, CRS query can also be useful in many real scenarios.

Modeling Imprecise User Intention. The clue is typically based on observations that 1) the keywords of PoIs in the clue may be interchangeable or inexact terms (e.g., a user may think of a PoI being canteen whereas it may be referred to as a restaurant); 2) the spatial relationships between PoIs are approximate, which is a natural phenomenon for human to estimate distance. For example, if the distance between two PoIs in the clue is about 100 meters, the actual distance may be noticeably greater than or less than 100 meters. Consider a scenario in our daily life: a user wants to find a restaurant in a city visited many years ago. She cannot remember the exact name and address but she still recalls that on the way driving to the restaurant from her home, she passed a cafe at about 1km away, and drove about another 2km to reach the restaurant. The information given above usually cannot precisely locate a PoI, but intuitively it provides clues to identify the most likely PoIs along the route.

Increased Flexibility in Trip Planning. As mentioned before, most existing work aims to find an optimal route with minimum travel distance. However, in many real scenarios, such an optimal route might not always be desirable. For instance, a user may have some personalized requirements on the distances between PoIs when planning a trip. Consider such a scenario, a user wants to find a buffet restaurant and a nearby cinema only in walking distance, say 3km, thus he can watch a movie after dinner. Therefore, after having delicious food, he can walk to the cinema in order to maintain a healthy lifestyle. These personalized requirements make the route search become distance-sensitive and more flexible such that the distance between PoIs along the route should be as close as possible to the user specified distance.

Clue-based Route Navigation. Given a description including textual and distance information on an expected route, it is still not direct-viewing enough for users to obtain the exact route. This is usually the case when a user wants to know the way for a specific place and asks the others for help, she may still not be able to exactly figure out the route after obtaining the answers from them, where the answer usually comes in the form, for example, "go straight on the way for about 100 meters, you will see a cafe, and turn right, you will arrive the Japanese restaurant after about 150 meters walk". Therefore, a novel type of route search which automatically interprets the clues contained in such answers becomes necessary. By augmenting it on current navigation services, a better user experience can be provided.

1.2 Challenge

In order to process the CRS query efficiently, we need to overcome several challenges. The first challenge is concerned with the large amount of possible routes for validation. Basically, the CRS requires candidate vertices that contain query keywords in the route to comply a specific order defined in query. As a feasible path is supposed to cover all the query keywords, the number of feasible paths increases exponentially with the amount of clues. Therefore, a greedy approach to solve our query is proposed, which continuously finds the next candidate vertex with minimum matching distance. Unfortunately, the optimal result can be substantially different from what the greedy algorithm suggests. Then, we propose a dynamic programming algorithm to answer CRS query exactly, but it requires quadratic time and is not scalable especially for more frequent keywords. To avoid unnecessary route search, we develop a branch-and-bound algorithm which adopts filter-and-refine paradigm, thus much fewer feasible paths are considered.

The second challenge is how to quickly locate candidate vertices in road networks. Given a query vertex u, the matching distance between u and its next candidate v is supposed to be smaller or equal to a threshold. The network expansion approach can be applied here, but it is inefficient due to excessive network traversals. Therefore, we propose a novel index structure, called AB-tree, which stores both keyword and distance information in each node. On top of it, the candidate w.r.t. a query clue can be quickly retrieved. The third challenge is how to reduce the index construction time and space. As AB-tree involves an all-pair matrix computation and has a space cost of O(|V | 2), we propose a PB-tree to further improve the performance. Inspired by the 2-hop label [1], [3], which answers distance queries with a small label index, we modify the structure of original label index to construct a binary tree on each pivot. In addition, we propose a semi-dynamic mechanism for PB-tree to support the index updating.

1.3 Contribution

The principal contributions of this paper can be summarized as follows.

• We propose a greedy clue search algorithm (GCS) to answer the CRS query approximately with no index involved. In GCS, we adopt the network expansion approach to greedily select the current best candidate at each step to construct feasible paths.

• We also develop a clue-based dynamic programming algorithm (CDP) that attempts to enumerate all feasible paths and finally returns the optimal result. In CDP, distance oracle is used to compute the network distance between candidates.

• We further propose a branch-and-bound algorithm (BAB) by applying filter-and-refine paradigm such that only a small portion of vertices are visited, hence improves

the search efficiency. In order to quickly locate the candidate vertices, we develop AB-tree and PB-tree structures to speed up the tree traversal, as well as a semi-dynamic index updating mechanism to keep the index maintainable when growing bigger.

• Our experimental evaluation demonstrates the efficiency of our algorithms and index structures for processing the CRS queries on real-world datasets. We show the superiority of our algorithms in answering CRS when compared with the baseline algorithms.

The remainder of this paper is organized as follows. We first formulate the problem of clue-based route search CRS in Section 2. Then we propose a greedy algorithm GCS in Section 3 to answer CRS approximately. Section 4 presents a clue-based dynamic programming algorithm CDP to return exact answer to CRS query. Efficient branch-and-bound algorithm BAB is introduced in Section 5, as well as two index structures ABtree and PB-tree. Section 6 presents a semi-dynamic mechanism for proposed index structure. Section 7 reports the experimental observations, and Section 8 reviews the related work. Finally, Section 9 concludes the paper.

II. PROBLEM STATEMENT

We model a road network as a weighted undirected graph G = (V, E), where V is the set of vertices and E is the set of edges. Each vertex v ∈ V contains a set of keywords, denoted as Φ(v). Each edge (u, v) ∈ E has a positive weight, i.e., length or travelling time on the edge, denoted as e(u, v). Given a path between vertices u and v, denoted as P(u, . . . , v), the length is the sum of weights of all edges along the path. For any two vertices u and v, the network distance between u and v on G, denoted as dG(u, v), is the length of the shortest path SP(u, v) between u and v. The notations used in this work is summarized in Table 1.

TABLE 1
Summary of notations

Notation	Definition
$G = (V, E)$	Road network with vertex V and edge E
$\mathcal{P}(u, \ldots, v)$	A path from u to v
$\mathcal{FP}(u, \ldots, v)$	A feasible path from u to v
$d_G(u, v)$	Network distance between u and v in G
$\mu(w, d, \epsilon)$	A clue with w, d and ϵ
$\sigma(u \rightarrow v)$	A match from u to v
$d_m(\mu, \sigma)$	Matching distance between μ and σ
$d_m(\mathcal{C}, \mathcal{FP})$	Matching distance between $\mathcal{C}$ and $\mathcal{FP}$
$L(v)$	2-hop label of v
$BT(v)$	Binary tree of v with keyword and distance
$PR(o)$	Pivot-based reserve label of vertex o
$PB(o)$	Binary tree of pivot o

2.1 Problem Definition

Definition 1 (Clue). A clue is defined as μ(w, d,), where w is a query keyword, d is a user defined distance, and ∈ [0, 1] is a confidence factor.

Definition 2 (Match). Given a source vertex u and a clue μ(w, d,), we say that the vertex pair σ(u → v) is a match w.r.t. clue μ, if the vertex v contains clue keyword w and the network distance between u and v is in [d(1−), d(1+)], i.e., w ∈ Φ(v) and dG(u, v) ∈ [d(1 −), d(1 +)].

Definition 3 (Feasible Path). We define a clue-based route query Q = (vq, C) where C is a sequence of clues denoted as C = {μ1(w1, d1, 1), . . . , μk(wk, dk, k)}. Given a query Q, if we find

(i) a sequence of vertices v0, v1, . . . , vk where v0 = vq and σ(vi−1 → vi) is a match w.r.t. μi for i ∈ [1, k]; (ii) a path P starts from v0 and passes v1, . . . , vk one by one;

(iii) Each subpath P(vi−1, . . . , vi) of P is the shortest path SP(vi−1, vi) for i ∈ [1, k].

We call such P a feasible path, denoted as FP(vq, v1, . . . , vk).

Definition 4 (Matching Distance). The matching distance between a clue μ(w, d,) and its match σ(u → v) in G, denoted as dm(μ, σ), is computed by d and the network distance dG(u, v), such that

$$dm(\mu, \sigma) = |dG(u, v) - d| \cdot d \quad (1)$$

It is worth noting that any monotonic increasing function that normalizes the matching distance into [0, 1] can be applied here.

The matching distance between C and its feasible path FP is defined as the maximum matching distance between all clues μ ∈ C and their corresponding matches σ ∈ FP, that is

$$dm(C, FP) = \max_{\mu i \in C, \sigma i \in FP} dm(\mu i , \sigma i) \quad (2)$$

The motivation of using Equation 2 is that the maximum matching distance of all the clues naturally controls the overall matching quality of the feasible path, which is a widely adopted w1 w2 w3 w4 w5 3 3 4 4 5 3 2 2 3 4 4 5 3 2 2 1 v1 v2 v3 v4 v5 v7 v8 v9 v6

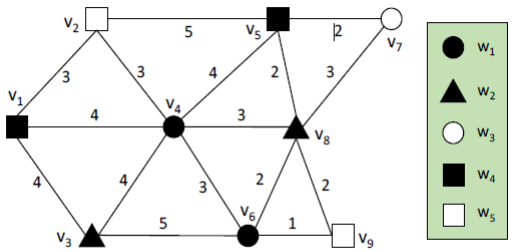


Fig. 1. Running example of G

method for the problem optimizing an objective score contributed by several components [15], [31].

Clue-based Route Search (CRS). Given a clue-based route search (CRS) Q = (vq, C), it aims to find a feasible path FP(vq, v1, . . . , vk), such that dm(C, FP) is minimized.

The clues in CRS implies that we are supposed to find a feasible path whose distances between two match vertices are as close as possible to user specified distance such that the user's search intention is satisfied. It is worth

noting that the CRS query can be easily extended to have a destination by assuming that the query keyword contained in destination is unique within G. In addition, for simplicity, we only discuss the optimal feasible path in this paper, but the algorithms introduced can be easily extended to find top-k feasible paths.

Example 1. Given Q = (v1, {(w2, 5, 0.5),(w1, 4, 0.5),(w3, 6, 0.5)}), thus both FP1(v1, v3, v6, v7) and FP2(v1, v3, v4, v7) are feasible paths with dm(C, FP1) = 0.4 and dm(C, FP2) = 0.5 respectively. Therefore, FP1 is reported as the result of CRS.

2.2 Preliminary: Distance Oracle

We adopt the idea of distance oracle DO to calculate the network distance between two input vertices. Given a source-target pair of vertices, DO returns the shortest network distance between them. As we know, the algorithms and data structures on DO have been extensively studied by existing works, which can be roughly summarized into two categories, expansion-based methods and lookup-based methods. The most famous expansion-based method for DO is Dijkstra's algorithm [14], which, given a s-t pair in road network G, traverses the vertices in G from s to t. However, the problem of using Dijkstra's algorithm is that it must visit every vertex that is closer to s, and the number of such unneeded vertices can be enormous when s and t are far apart, which incurs redundant network traversal. Besides, the lookup-based methods usually have to store some precomputed results. For example, allpair method is space inefficient that we have to precompute and store a distance matrix, which requires O(n 2) space for a road network G with n vertices. To the best of our knowledge, one of the most notable recent developments is the emergence of practical 2-hop labeling methods [1]–[3], [18] for DO on large networks. It constructs labels for vertices such that a distance query for any vertex pair u and v can be answered by only looking up the common labels of u and v. For each vertex v, we precompute a label, denoted as L(v), which is a set of label entries and each label entry is a pair (o, ηv,o), where o ∈ V and ηv,o = dG(v, o) is the distance between v and o. We say that o is a pivot in label entry if (o, ηv,o) ∈ L(v). Given two vertices u and v, we can find a common pivot o that (o, ηu,o) ∈ L(u) and (o, ηv,o) ∈ L(v): dG(u, v) = min{ηu,o + ηv,o} (3) We say that the pair (u, v) is covered by o and the distance query dG(u, v) is answered by o with smallest ηu,o + ηv,o. Therefore, we can compute dG(u, v) in O(|L(u)| + |L(v)|) time by using a merge-join like algorithm.

III. GREEDY CLUE SEARCH ALGORITHM

We develop a greedy algorithm as a baseline for answering the CRS query, which is called Greedy Clue Search (GCS) algorithm. Given a query Q = (vq, C), we first add vq into a candidate path. Then we use the Procedure findNextMin () to determine the next match vertex v1 that the matching distance between μ1 and σ1(vq → v1), i.e., dm(μ1, σ1), is minimized. Afterwards, we insert v1 into the candidate path, and continue to find its contagious candidate by findNextMin (). This process is repeated until all the match vertices are determined, thus the candidate path forms a feasible path, denoted as FPvq If we assume Procedure findNextMin() costs time f, then the time complexity of GCS is O(k · f).

In Procedure findNextMin(), we utilize the network expansion algorithm [16] to find the nearby vertices that contain the query keywords and the network distances are in the confidence intervals. The algorithm details are shown in Algorithm 1. Given the source u, and the clue μ(w, d,), we aim to find a match vertex v such that the difference between dG(u, v) and d is minimized. In the network traversal starting from u, we check every visited vertex to see if it is a match vertex that contains w and locates in the interval [d(1 −), d(1 +)]. If v is the first visited match vertex and dG(u, v) > d, then we stop and return v since the difference incurred by the remaining unvisited vertices cannot be less than dG(u, v) − d. Otherwise, we continue to find the next match vertex v 0 . If v 0 is found, then

(i) If dG(u, v0) ≤ d, we update v by v 0 since v 0 renders a smaller difference than v;

(ii) Otherwise, we compare d − dG(u, v) with dG(u, v0) − d and return the smaller one as the result.

Example 2. In running example Figure 1, we are given CRS query Q = (v1, {(w2, 5, 0.5),(w1, 4, 0.5),(w3, 6, 0.5)}). First, we fetch v1 into the candidate path, and call findNextMin(v1, w2, 5, 0.5) and return v3 with dm = 0.4. Therefore, we repeat the process and finally obtain FPv1 = (v1, v3, v4, v7) with dm(C, FPv1) = 0.4. Therefore, we have FPgcs = FPv1 and dm(C, FPgcs) = 0.4.

IV. CLUE-BASED DYNAMIC PROGRAMMING ALGORITHM

As we know, even though GCS has a short response time, the accuracy of the answer cannot be guaranteed. To achieve better accuracy, we propose an exact algorithm, called Clue-based Dynamic Programming (CDP), to answer the CRS query. Generally, it is challenging to develop an efficient exact algorithm for CRS queries, since we cannot avoid exhaustive search for PoIs in road networks. For instance, the number of vertices that contain keyword wi ∈ C is denoted as |Vwi |, thus the time complexity of the

Algorithm 1: Procedure find Next Min ()

Input: Source vertex u and clue μ(w, d,)
Output: min {dm (μ, σ)} and match vertex v
1 From u, do network traversal;
2 if a match vertex v is found then

3 dG ← the network distance between u and v;
4 while true do
5 Find next v 0 contains w, thus obtain d 0 G;
6 if dG < d and d 0 G > d then
7 break;
8 else
9 v ← v 0 and dG ← d 0 G;
10 return min{dm(μ, σ)} and v;

brute-force approach, which attempts all possible combinations, is O(Q wi∈C |Vwi |). In CDP, we construct a keyword posting list for each keyword w, which is a list of vertices that contain w. When a CRS query is issued, we sort the posting lists according to the keyword order of wi ∈ C. Note that the order of the vertices within each posting list does not matter and can be arbitrary, hence are sorted by vertex id for simplicity. It is easy to see that these posting lists actually construct a k-bipartite graph G0 , which in fact shows all feasible paths for a given C. The weight of each edge in G0 is computed as the matching distance. Specifically, for each u ∈ Vwi , we define D(wi , u) to denote the minimum matching distance one can achieve with a walk that passes the keywords from w1 to wi consistent with the order in C and stops at u. In other words, the weight of vertex u ∈ G0 is computed by D(wi , u), which is the minimum matching distance of all partial feasible paths end at u. Then we compute D(wi , u) by the following recursive formula:

(i) i = 1: for match vertices u ∈ Vw1 , we have D(wi , u) = dm(μi(wi , di), σ(vq → u))

(ii) i > 1: for match vertices v ∈ Vwi−1 and u ∈ Vwi , we have D(wi , u) = min v∈Vwi−1 {max{D(wi−1, v), dm(μi , σ(v → u))}} (4)

(iii) For each iteration, we have at most |Vwi−1 | · |Vwi | combinations, thus the time required is O(Pk i=2 |Vwi−1 | · |Vwi |). The details of CDP are shown in Algorithm 2. In order to compute D(wi , u), we have to access the posting list of wi−1. For each vertex v in this list, we first check if σ(v → u) is a match w.r.t. μi and then compute dm(μi , σ(v → u)). Then we compare it with D(wi−1, v), and keep the greater one as intermediate value. Finally, we find the minimum one as D(wi , u) from these |Vwi−1 | intermediate values. After we recursively process all the keywords, we finally find the minimum D(wk, u) and back trace the corresponding vertices that construct FP cdp. In each iteration, we have a clue μi(wi , di), therefore we have to compute dG(u, v) between each u ∈ Vwi and its precedents v ∈ Vwi−1 as prerequisites for determining dm(μi , σ(v → u)). Here we adopt the distance oracle introduced in Section 2.2 to compute dG(u, v).

Example 3.
As shown in Figure 2, given CRS query Q = (v1, {(w2, 5, 0.5),(w1, 4, 0.5),(w3, 6, 0.5)}). To compute D(w3, v7), we first compare D(w1, v4) = 0.4 with dm(μ3, σ(v4 → v7)) = 0, and obtain intermediate

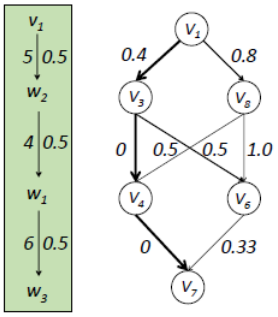


Fig. 2. Matching distances of CDP

Likewise, we have D(w1, v6) = 0.5 and dm(μ3, σ(v6 → v7)) = 0.33, thus the intermediate value is 0.5. Therefore, CDP returns FPcdp = (v1, v3, v4, v7) with dm(C, FPcdp) = 0.4.

Algorithm 2: Clue-based Dynamic Programming CDP Input: Q = (vq, C = {(w1, d1), . . . ,(wk, dk)}) Output: FPcdp with dm(C, FPcdp)

1 for each u ∈ Vw1 do
2 Initial D(w1, u);
3 for 1 < i ≤ k do
4 for each u ∈ Vwi do
5 Initial intermediate vector iv(u);
6 for each v ∈ Vwi−1 do
7 if dm(μi, σ(v → u)) < D(wi−1, v) then
8 iv(u) insert D(wi−1, v);
9 else
10 iv(u) insert dm(μi, σ(v → u));
11 D(wi, u) ← min{iv(u)}
12 Find min{D(wk, u)};
13 return FPcdp and dm(C, FPcdp) ← min{D(wk, u)};

V. BRANCH AND BOUND ALGORITHM

Although CDP provides an exact solution, the search efficiency cannot be maintained. For instance, consider the worst case, we assume that all vertices contain query keywords, then the time is O(k · |V | 2). To propose a more efficient algorithm, we assume there is an artificial directed graph G0 , which is similar to the k-bipartite graph in CDP that formed by all candidate vertices containing keywords in C, where the edge of G0 is a match of one clue and in the meantime its direction complies the keyword order of the clue. Note that, G0 is organized into k levels, and each level i corresponds to each keyword wi . Based on G0 , we develop a Branch-and-Bound (BAB) algorithm to search G0 in a depth-first manner by applying the filter-and-refine paradigm, which only visits a small portion of vertices in G0 . Fortunately, we can use the result of GCS to speed up the search process since it can serve as an initial upper bound.

5.1 Algorithm Outline

We start the searching from level 1 to k to obtain a feasible path FP, if the matching distance dm(C, FP) is greater than the current upper bound, we continue to search for the next candidate feasible path, otherwise we update the upper bound. It is worth noting that it is not necessary to go through every candidate feasible path. If the matching distance at intermediate level already exceeds the upper bound, it can be removed. This process terminates when the matching distance next to be processed at level 1 can be filtered, since it is impossible to find a feasible path with smaller match distance.

Candidate feasible path updating. Initially, we keep a stack to store the partial candidate path, which contains a sequence of vertices and corresponding matching distances. First, we fetch vq into the stack, then we continue to find next candidate at level 1. Basically, the key component of this algorithm is to quickly locate the next best match vertex, and the details of Procedure findNext () will be introduced later. Given a partial candidate path FP (vq, v1, vi) obtained at level i, we apply findNext () to find the next candidate vi+1 at level i+ 1. Once vi+1 is found, we compute d i+1 m (vi+1) which denotes the matching distance at level i + 1 resulted by vi+1, and compare it with current UB. Note that, vi+1 is accepted as a candidate and inserted into the stack if and only if its matching distance d i+1 m (vi+1) is smaller than UB. Otherwise, vi is removed from the stack as well as d i m(vi). In other words, vi is not valid that the path FP(vq, v1, . . . , vi−1) cannot survive by passing vi , then we have to find an alternative v 0 i . As we know vi is the current best candidate at level i, therefore we have to relax the matching distance by finding v 0 i where d i m(vi) ≤ d i m(v 0 i) and d i m(v 0 i) is minimum among all the rest vertices untouched at level i. Afterwards, if v 0 i is valid, we continue to apply findNext() on it.

Upper bound updating. Specifically, after we obtain a feasible path FP(vq, v1, . . . , vk−1) at level k − 1, if vk is returned by findNext(), then we check if d k m(vk) exceeds UB. If vk is not valid, we prune vk and simply repeat the above process. Otherwise, we insert vk into the stack, and a complete feasible path is determined. Hence, FP(vq, v1, . . . , vk) is regarded as a temporary result, and UB is updated by the minimum matching distance among all d i m(vi)s. It is easy to see that, we cannot find a better feasible path by alternating vk with v 0 k at level k, since no further level is available to make up the relaxation caused by v 0 k . Therefore, in addition to remove vk, we continue to remove vk−1 from the stack and repeat the above process. In general, the pruning happens from the lower levels to the higher levels, i.e., from level k to level 1. In the end, at level 1, if the matching distance induced by the next candidate vertex is greater than UB, it is impossible to find another feasible path, thus the stack becomes empty after the last vertex vq is removed, and this process terminates.

Example 4. In the running example, given query Q = (v7, {(w1, 6, 0.5),(w2, 4, 0.5),(w4, 5, 0.5)}). First we fetch v7 into the stack, and findNext() returns v4 with d 1 m(v4) = 0. Then we insert v4 into stack and continue to find next candidate vertex, and v3 is obtained with d 2 m(v3) = 0. The process continues and then we have v1 with d 3 m(v1) = 0.4. As the size of stack is same as the number of query keywords, a feasible path FP = (v7, v4, v3, v1) with dm(C, FP) = 0.4 is obtained, and UB is updated by 0.4. Next, we remove v1 and v3 from the stack, and continue to find next candidate of v4. As d 2 m(v3) = 0, we relax the matching distance and call findNext() which returns v8 with d 2 m(v8) = 0.5. Then we have to remove v4 from the stack since d 2 m(v8) already exceeds current upper bound UB. Now we move on to apply findNext() on v7 and returns v6 with d 1 m(v6) = 0.33. However, the next candidate v3 has d 2 m(v3) = 0.5 greater than UB, thus we remove v6 and v7 from stack. Therefore, the algorithm terminates since no other feasible path exists. We have FPbab = (v7, v4, v3, v1) with dm(C, FPbab) = 0.4.

Algorithm 3: Branch and Bound BAB
Input: Q = (vq, C)
Output: FPbab with dm(C, FPbab)
1 Initialize stackV, stackD, and search threshold θ;
2 Push vq into stackV;
3 while stackV is not empty do
4 i ← stackV.size();
5 if findNext(vi−1, di, wi, θ) = true then
6 Obtain vi and d i m(vi);
7 θ ← 0.0;
8 Push vi into stackV and d i m(vi) into stackD;
9 if i equals to k then
10 if max{stackD} <= UB then
11 Update UB by max{stackD};
12 Update FPbab by stackV;
13 Update θ by top of stackD;
14 Update stackV and StackD;
15 else 16 Update θ by top of stackD;
17 Update stackV and StackD;
18 return FPbab and dm(C, FPbab) ← UB;

5.2 All-Pair Distance Approach

In BAB, the Procedure findNext() is applied on vi−1 to find the next candidate vertex vi . We can simply use Procedure findNextMin() in GCS to locate the next candidate, but it is inefficient due to redundant network traversal especially when di ∈ μi is large. Moreover, when we prune vi and attempt to find alternative v 0 i , it is easy to see findNextMin() cannot be directly applied. Therefore, we propose an All-pair Binary tree (AB-tree) index to improve the search efficiency.

5.2.1 All-Pair Binary Tree

Given a vertex u, we aim to find a vertex v containing keyword w such that the matching distance between σ(u → v) and query clue μ is slightly greater than and closest to a threshold θ among all vertices containing w in G. Note

that, the threshold θ is settled by previous filtered candidate at the same level with v, and it is 0 at initial stage. In other words, we are supposed to find the vertex v that the difference between dG (u, v) and d ∈ μ is close to θ · d. To this end, we construct AB-tree as follows.

For each v ∈ V, we construct a binary tree BT (v) that contains the information of network distances and keywords. After the all-pair distance matrix is obtained, for each v, we have a list of vertices sorted in ascending order of network distance to v. By utilizing the tree structure, the vertices in the list are divided into fragments that the network distances w.r.t. v of the vertices in the same fragment are close to each other, which speeds up the looking up for vertices by network distance. In addition, the keyword information within each fragment is also stored in BT(v) such that the vertices containing query keyword in a fragment can be efficiently retrieved.

We utilize a hash function H that maps keywords and vertices to a binary code with h bits. For each keyword w, one of its h bits in H(w) is set to 1. Hence, the binary code of a vertex v is

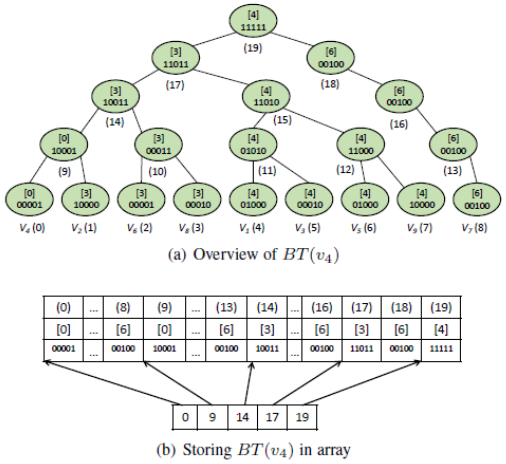


Fig. 3. Overview of all-pair binary tree, with illustration of the two components: (a) Example of $BT(v_4)$; (b) Storing in array.

the superimposition of H(v) that v ∈ S. It is worth to note that a non-zero value of H(w) ∧ H(S) indicates that there may exist a vertex v ∈ S containing w, and H(w) ∧ H(S) = 0 means w is definitely not contained by any v ∈ S. BT(v) is actually a B+- tree with fanout f = 2. Each leaf node contains the information of a vertex u with both the network distance dG(u, v) and binary code H(u) stored. For a non-leaf node, it also keeps a routing element, which equals the maximum network distance of its left subtree. Therefore, BT(v) is constructed recursively in bottom-up manner as shown in Figure 3(a).

Storing BT(v) in an array. As we know, storing the tree structure as an array enables a better performance than storing pointers. Therefore, we propose a scheme to sequentially store all nodes of BT(v) in an array from nodes on height 0 to the root, as shown in Figure 3(b). In addition to this array, we also keep an auxiliary array that indicates the number of nodes in each level of BT(v), by which we can quickly determine the indices of the subnodes of a non-leaf node, or the index of its parent node, in the array. For example, if we want to find the left and right subnodes of node 16 in BT(v4), we know there are two nodes on its left side by 16− 14 = 2 where 14 is the start index of nodes at height 2, so the index of its left subnode is 9 + 2 ∗ 2 = 13 and the right is 9 + 2 ∗ 2 + 1 = 14. However, we notice 14 is actually at height 2, then we figure out node 16 does not have a right subnode.

Lemma 1. Given G = (V, E), the space cost of AB-tree is O(|V | 2 · h).

Proof. For each v ∈ V , we have |V | elements in distance matrix, thus each BT(v) has an index size O(|V | · h). It is easy to see the size of AB-tree is O(|V | 2 · h).

5.2.2 Predecessor and Successor Queries on AB-tree

After the construction of AB-tree, we discuss how to use it so that the next vertex in candidate path can be quickly located. Initially, if there is no previous vertices accessed at the next level of vi−1, network distance dG(vi−1, vi) between vi−1 and next candidate vi is supposed to be smaller or equal to lD = di , or greater or equal to rD = di , where di ∈ μi . Additionally, consider the aforementioned scenario, we have FP(vq, v1, . . . , vi), but vi+1 returned at level i+ 1 exceeds UB. Then we have to remove vi from the stack and turn to find v 0 i as alternative, where d i m(vi) ≤ d i m(v 0 i). It is easy to see the difference between dG(vi−1, v0 i) and di must be greater or equal to d i m(vi) · di . In other words, the network distance dG (vi−1, v0 i) is smaller or equal to lD or greater or equal to rD, where

lD = di − d i m(vi) · di ,
rD = di + d i m(vi) · di ,

we have dG(vi−1, v0 i) ≤ lD or dG(vi−1, v0 i) ≥ rD.

(5) Therefore, the predecessor and successor queries can be issued on BT(vi−1) to retrieve next candidate with two boundary network distances lD and rD, respectively.

Predecessor query. Given BT(u), a query keyword w and network distance lD, we aim to find vertex v that contains w and dG(u, v) is smaller or equal to and closest to lD. First, we compute binary code H(w) for query keyword w. Then we start the process of searching BT(u) recursively from top to bottom. For a non-leaf node o, if H(w) ∧ H(o) is non-zero, we continue to search its subtrees. If lD is smaller than the routing element of o, only its left subtree needs to be considered. Otherwise, we first check if we could find v in its right subtree (if exists), if not, we turn to search its left subtree. For a leaf node v, we directly check if v contains w and dG(u, v) is smaller or equal to lD, therefore, false positives can be avoid. Finally, v is obtained. For example, a predecessor query on BT(v4) with keyword w2 and lD = 4. First, we have H(w2) = 00010. The search starts from root, and as lD equals to the routing element 4, thus we first search its right subtree. After checking H(w2) with binary code of Node 18, we find it does not contain w2 and we turn to search Node 17.

As the routing element of Node 17 is smaller than lD, we move to search Node 15. Then we check H(w2) with the binary code in Node 12, and find Node 12 does not contain w2. After checking with Node 11, we have v3 as result of the predecessor query.

Successor query. Likewise, we have BT(u), a query keyword w and network distance rid, the goal is to find vertex v that contains w and dG(u, v) is greater or equal to and closest to rD. For a non-leaf node o, if H(w) ∧ H(o) is non-zero, the subtrees of o need to be considered. If rD is smaller or equal to the routing element of o, we search the left subtree to see if v could be found, if not, we turn to search the right subtree (if exists). Otherwise, we simply search the right subtree (if exists) to locate v. For a leaf node v, if v contains w and dG(u, v) is greater or equal to rD, v is reported as result. For example, a successor query on BT(v4) with keyword w2 and rD = 4. We first check the root with H(w2) = 00010, and rD equals to the routing element, which means we first search Node 17 to see if it contains w2, then search Node 18. As the routing element of Node 17 is smaller than rD, we only need to check Node 15. Then, since the routing element of Node 15 is same as rD, we turn to search Node 11. Finally, we obtain v3 as result of the successor query.

As mentioned before, we process a predecessor and a successor queries on BT(vi−1) with lD and rD respectively to locate candidate at level i. If both predecessor and successor queries find candidate vertices, we compare their matching distance and report the smaller one as result. If only one of them finds candidate vertex, we directly report it. Otherwise, no candidate is found. Note that, in the process to replace vi with v 0 i , we must skip vi in the tree traversal to avoid infinite loop caused by the special case d i m(vi) = d i m(v 0 i).

Lemma 2. The expected number of nodes visited in a predecessor or successor query on AB-tree is O(log |V |·|W|·|Φ(V)| |V |·h·|Vwk |).

Proof. Assume W is the keyword set of G and the keywords are evenly distributed. The hash function maps each keyword w ∈ W to a binary code with h bits based on its modulus, which might lead to false positives. We denote the average number of conflicting keywords as |W| h and the average keyword frequency as favg = |Φ(V)| |V | . Therefore, the probability of false positive can be computed as p = 1 − |V |·h·|Vwk | |W|·|Φ(V)| . When encountering a false positive at the leaf node, the predecessor query traces back from the right subtree and continues to search the left subtree to find another leaf node. In this case, O(log |V |) tree nodes will be visited for each trace back at the worst case. The algorithm terminates when it reaches a true positive. Hence, the expected number of trace backs is equivalent to the expected number of false positives before a true positive, which is 1 1−p = |W|·|Φ(V)| |V |·h·|Vwk | . In total, the expected number of tree nodes visited in a predecessor query is O(log |V |·|W|·|Φ(V)| |V |·h·|Vwk |). Assume the time cost for ∧ operation on two h length hash codes is O(h), thus the time of searching ABtree is O(log |V |·|W|·|Φ(V)| |V |·h·|Vwk |) · O(h).

Example 5. In the running example, given query Q = (v7, {(w1, 6, 0.5),(w2, 4, 0.5),(w4, 5, 0.5)}), assume we already have stack (v7, v4, v3). At level 2, we intend to remove v3 and find an alternative. Given d 2 m(v3) = 0, we apply a predecessor and successor queries on BT(v4). For the predecessor query, we take (w2, 4, 0.5) and 0.0 as input. As v3 is previous result, we skip it and return v8. For the successor query, no vertex is found. Therefore, we report v8 as our next candidate with d 2 m(v8) = 0.5. 5.3

5.3 Keyword-based Label Approach

Even though AB-tree is able to answer findNext() query fast, the index space cost is still high and could only be stored in disk, which results in undesired I/O consumption. In this section, we introduce a main memory based index structure, namely Pivot reverse Binary tree (PB-tree), to deal with findNext() query.

5.3.1 Pivot Reverse Binary Tree

As introduced in Section 2.2, we know 2-hop label possesses the nature to process distance queries between any two vertices in network with fast response time, whilst keeping the size of the generated label index as small as possible. The problem of reducing label size is orthogonal to our work, thus we fully utilize the stateof-the-art results to build a small index in this work. As we know, in 2-hop label, the distance between any vertex pair (u, v) can be computed correctly through a common pivot o, in other words, each vertex u can reach any other vertex v in network through a pivot o. Therefore, based on this intuition, we modify the structure of original 2-hop label to construct a pivot reverse index, i.e., P R index [34] which stores all label entries (o, ηv,o) ∈ S v∈V L(v) regarding vertex o as pivot into the P R label of vertex o, i.e., (v, ηv,o) ∈ P R(o). In P R(o), we assume that all the label entries (v, ηv,o) are sorted in ascending order of distance.

Algorithm 4: Procedure findNext() with AB-tree Input: Query vertex vi−1, clue wi and di, threshold θ Output: Next candidate vi with d i m(vi)

1 Obtain BT(vi−1);
2 lD ← di − di · θ; rD ← di + di · θ;
3 vp and dp ← BT(vi−1).predecessor(lD, wi) ;
4 vs and ds ← BT(vi−1).successor(rD, wi) ;
5 if di − dp ≤ ds − di then 6 return vp with dm(vp); 7 else
8 return vs with dm(vs); Procedure Predecessor(lD, w, Node) 1 if Node is a leaf node then 2 Obtain vp and dp of current node; 3 if vp contains w and dp ≤ lD then 4 return vp and dp; 5 else 6 return false; 7 else 8 Generate H(w);
9 if H(w) ∧ H(Node) = 0 then 10 return false;
11 if lD < Node.routing then
12 lNode ← index of its left subnode;
13 return Predecessor(lD, w, lNode);
14 else
15 rNode ← index of its right subnode;

16 lNode ← index of its left subnode;
17 if rNode exists then
18 if Predecessor(lD, w, rNode);
19 then
20 return vp and dp
21 else
22 return Predecessor(lD, w, l Node);
23 return Predecessor(lD, w, l Node); example, we have (v3, 0) ∈ L(v3) and (v3, 4) ∈ L(v1). Through the transformation, we have P R(v3) = {(v3, 0),(v1, 4)}. In order to find vertex by keyword and distance information, each P R(o) is organized as same as the binary tree mentioned before, thus forms P B(o). The structure is shown in Figure 4, it is worth to note that any network distance dG(u, v) is divided into two parts, the first part dG(u, o) between u and its pivot o can be found in L(u), and the other part dG(o, v) between pivot o and target v can be found in P B(o). Therefore, combined with original label index whose label entries are also sorted in ascending order by network distance, PB-tree could be used to answer predecessor and successor queries more efficiently than AB-tree with a much smaller size.

Lemma 3. Given G = (V, E) and label index L(v) for all v ∈ V , the space cost of PB-tree is O(|L| · h). Proof. For each v ∈ V , we have |L(v)| label entries, thus each P B(v) has an index size O(|L(v)| · h). It is easy to see the size of PB-tree is O(|L| · h) where |L| is the size of label index.

5.3.2 Predecessor and Successor Queries on PB-tree With the construction of PB-tree, we discuss the predecessor and successor queries on top of it. Given P B(vi−1), we aim to find candidate vi that contains wi and dG(vi−1, vi) is smaller or equal to lD, or greater or equal to rD. As we know, dG(vi−1, vi) can

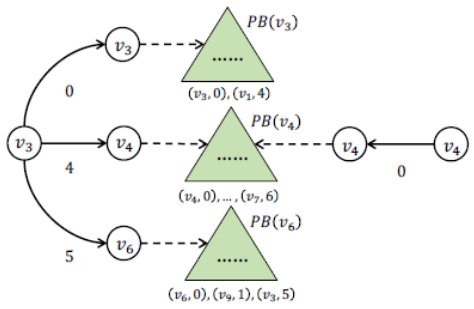


Fig. 4. Overview of pivot reverse binary tree

be divided into two parts dG(vi−1, o) and dG(o, vi). Therefore, straightforwardly, we can apply predecessor and successor queries on P B(o) for each pivot o ∈ L(vi−1) with two bound network distances lDo and rDo, respectively. Therefore, for dG(o, vi), lDo = lD − dG(vi−1, o),
rDo = rD − dG(vi−1, o),
we have dG(o, vi) ≤ lDo or dG(o, vi) ≥ rDo. (6)

For each P B(o), we are supposed to obtain a temporary candidate. Through comparison, we can finally find the next candidate vertex vi .

Fortunately, it is worth to note that we are not necessarily to access all P B(o)s to process predecessor and successor queries. Basically, we know d i m(vi) must not exceed upper bound matching distance, therefore current UB can be utilized to prune the search space. That is to say, vi could only be found if dG(vi−1, vi) is greater or equal to lB, or is smaller or equal to rB.
lB = di − di ∗ UB,
rB = di + di ∗ UB,
we have dG(vi−1, vi) ≥ lB or dG(vi−1, vi) ≤ rB. (7) Particularly, for each P B(o), the bound distances can be computed as
lBo = lB − dG(vi−1, o),
rBo = rB − dG(vi−1, o). (8)
Therefore, the search space can be narrowed down into [lBo, lDo] and [rDo, rBo]. For current pivot o being processed, if we have rB < dG(vi−1, o), we are impossible to find a candidate in P B(o) since rBo is negative. In other words, the network distance between vi−1 and any vertex in P B(o) is definitely greater than rB thus is not qualified. As we know, the pivots in L(vi−1) are sorted in ascending order of network distance, the rest pivots o 0 after o do not need to be considered since they have even greater network distances to vi−1 than o. Therefore, the process terminates.

Predecessor and successor queries. Given P B(o), a query keyword w and two network distance bound ranges [lBo, lDo] and [rDo, rBo], we aim to find a temporary candidate vertex in P B(o). In particular, the difference between AB-tree and PBtree is that, given a query vertex u, any target v only shows up once in AB(u), but it might appear in multiple P B(o)s. Moreover, if we find a candidate v in P B(o), dG(u, o)+dG(o, v) is not necessarily equal to dG(u, v) since the network distance can only be calculated by the pivot with minimum distance summation. Therefore, we use original label index to check if P(u, . . . , o, . . . , v) is the shortest path SP(u, v). As mentioned before, if rB ≥ dG(vi−1, o), we first apply a successor query on P B(o). After we obtain a temporary vertex vtmp locates in [rDo, rBo], we check if o is on the shortest path SP(vi−1, vtmp) by comparing dG(vi−1, vtmp) with dG(vi−1, o) + dG(o, vtmp). If so, vtmp is reported as a temporary successor result on P B(o). Otherwise, we update rDo by dG(o, vtmp) and continue to apply a new successor query. This process is repeated until we find a result. After successor query, we compare dG(vi−1, o) with lD to determine if we need to apply a predecessor query on P B(o). Based on the same intuition, if lD ≥ dG(vi−1, o), the predecessor query is applied in a similar approach as successor query. Finally, we compare the results of predecessor and successor queries, and obtain the temporary candidate found in P B(o). It is worth to note that we can further narrow down the search space by

updating lB and rB. That is, after processing pivot o, if we find a temporary candidate vtmp, lB can be updated by dG(vi−1, vtmp) and rB by 2 ∗ di − lB, which benefits the processing of rest o 0 .

Lemma 4. The expected number of nodes visited in a predecessor or successor query on PB-tree is O(log |L| |V | · |W|·|Φ(V)| |V |·h·|Vwk |).

Proof. According to the proof in Lemma 2, the expected number of false positives before a true positive is 1 1−p = |W|·|Φ(V)| |V |·h·|Vwk | . As we know, the height of each PB-tree is log |L| |V | . Therefore, expected number of nodes visited in a predecessor or successor query on PB-tree is O(log |L| |V | · |W|·|Φ(V)| |V |·h·|Vwk |). Given that ∧ operation on two h length hash codes costs O(h), thus the time of searching PB-tree is O(log |L| |V | · |W|·|Φ(V)| |V |·h·|Vwk |) · O(h).

Example 6. In the running example, given query Q = (v7, {(w1, 6, 0.5),(w2, 4, 0.5),(w4, 5, 0.5)}), assume we already have stack (v7, v4, v3). At level 2, we intend to find the next candidate. Initially, θ is set as 0.0, therefore we have lD = rD = 5. As current UB = 0.4, we have lB = 4 and rB = 6. As shown in Figure 4, we first check P B(v3) with dG(v3, v3) = 0. Then we have lDv3 = rDv3 = 5, lBv3 = 4 and rBv3 = 6. A successor query is applied and no vertex is found, and a predecessor query returns v1. As dG(v3, v1) = 4 does not exceed lBv3 , v1 is taken as the temporary result for pivot v3. Then we continue to search P B(v4) with lDv4 = rDv4 = 1, lBv4 = 0 and rBv4 = 2 but no vertex is found, neither in P B(v6). Finally, we report v1 with d 3 m(v1) = 0.4.

VI. DYNAMIC MAINTENANCE

In this section, we discuss how to maintain the PB-tree for road network updating. To avoid recomputing the index structure from scratch, we propose a semi-dynamic mechanism to adjust the PBtree with a low overhead. As we know, PB-tree is built based on label index, thus the updating is divided into two phases, the updating of label index and the updating of PB-tree. Instead of recomputing a new label index, [4] introduces a dynamic label index scheme for distance queries on time-evolving graphs, and we adopt the algorithm for the first phase label index updating.

Algorithm 5: Procedure findNext() with PB-tree Input: Query vertex vi−1, clue wi and di, threshold θ Output: Next candidate vi with d i m(vi)

1 lD ← di − di · θ; rD ← di + di · θ;
2 lB ← di − di · UB; rB ← di + di · UB;
3 for each pivot o ∈ L(vi−1) do
4 Obtain P B(vi−1), lDo, rDo, lBo and rBo;
5 if dG(vi−1, o) > rB then
6 break;
7 else
8 rDo ← rD − dG(vi−1, o);
9 while P B(vi−1).suck(rDo, wi) and dG(o, vtmp r) ≤ rBo do
10 Obtain vtmp r;
11 if dG(vi−1, vtmp r) 6= dG(vi−1, o) + dG(o, vtmp r) then
12 rDo ← dG(o, vtmp r);
13 else
14 Obtain temp suc result on P B(o);
15 break;
16 if dG(vi−1) < lD then
17 lDo ← lD − dG(vi−1, o);
18 while P B(vi−1).pred(lDo, wi) and dG(o, vtmp l) ≥ lBo do
19 Obtain vtmp l;
20 if dG(vi−1, vtmp l) 6= dG(vi−1, o) + dG(o, vtmp l) then
21 lDo ← dG(o, vtmp l);
22 else
23 Obtain temp pre result on P B(o);
24 break;
25 if di −dG(vi−1, vtmp l) ≤ dG(vi−1, vtmp r)−di then
26 lB ← dG(vi−1, vtmp l); rB ← 2 ∗ di − lB;
27 vi ← vtmp l;
28 else
29 rB ← dG(vi−1, vtmp r); lB ← 2 ∗ di − rB;
30 vi ← vtmp r; 31 return vi with d i m(vi); 6.1

Semi-Dynamic Index Structure

Basically, we have 4 operations to update the network: insert a new vertex with an edge connecting to an existing vertex, delete a vertex with only one edge, insert an edge and delete an edge. As the deletion operation is much harder than insertion, and it seems impossible to find an efficient approach to support deletion in label generation. Moreover, it is rare to see deletion happens in road networks, thus we only take insertion into consideration. As the newly updated vertex is isolated, its label can be viewed as an empty set. Inserting a new vertex can be easily done by inserting an edge connecting to it, thus we only need to focus on edge insertion. As keyword updating is easy to implement, thus we omit it here.

Label index updating. Assume we insert an edge (a, b) into G, some shortest paths in old network may change by passing (a, b). Based on the label generation algorithm, we do not have to remove outdated distances in label but resume BFSs of affected vertices and add new label entries into index. It is worth to note that only the pivots in L(a) and L(b) are affected by network updating, and it suffices to conduct resumed BFSs originally rooted at pivot vk if vk ∈ L(a) ∪ L(b). Different with previous pruning method, a prefixal pruning method is proposed to apply in BFS with a new parameter k, where k is the vertex ordering of vk. The prefixal method is to answer the distance query between vk and u from the pivots in L(vk)∩L(u) whose vertex orderings are at most k. Interested readers can refer to [4] for algorithm details.

Pivot-based forest. To propose a semi-dynamic index structure, we present a general framework to convert

PB-tree into pivot-based forest (PF), which is inspired by the logarithmic method [5]. Given P B(o) with m label entries, we divide it into l = blog mc + 1 partitions P0, . . . Pl−1. Each partition Pi either has 2 i label entries or is empty. We first compute a l-bit binary value of m. Interestingly, whether Pi is empty or not is determined by the ith bit, if ith bit is 0 then Pi is empty. For non-empty Pi , we follow the method introduced in Section 5.2.1 to construct a binary tree P F(o)i on these 2 i label entries. Finally, all these binary trees together form the pivot-based forest structure.

PF index updating. After label index updating, we add new label entries or rewrite distances of existing label entries. Assume we add a new label entry (v, dG(o, v)) into P B(o), we first find the smallest i such that P F(o)i is empty. If i equals to 0, we simply build P F(o)0 with only one label entry (v, dG(o, v)). Otherwise, we union all label entries of P F(o)o, . . . , P F(o)i−1, together with (v, dG(o, v)), into P F(o)i . It is worth to note that P F(o)i now has 2 i elements and P F(o)o, . . . , P F(o)i−1 become empty. As we know, the label entries in original P B(o) are sorted in ascending order of distance. In P F(o), we do not consider the global distance order but instead consider a local order in each P F(o)i when we rebuild the index. To rewrite distances of existing label entries, we only need to update the P F(o)i they belong to.

Query processing on PF index. Given query vertex vi−1 and a clue µ(wi , di), we introduce how to answer findNext() on PF index. As we know, both the predecessor and successor queries are decomposable. Therefore, we simply apply the predecessor and successor queries on all non-empty P F(o)i . Fortunately, it is not necessary to process queries on all P F(o)is. If the query distance is smaller than the minimum network distance stored in P F(o)i , the predecessor query is not required, where the similar case holds for successor query. Finally, we merge these intermediate results to obtain the result.

VII. EXPERIMENTS

In this section, we conduct extensive experiments on real road network datasets to study the performance of the proposed index structures and algorithms.

7.1 Experimental Settings All these algorithms introduced in this paper were implemented in GNU C++ on Linux and run on an Intel(R) CPU i7-4770@3.4GHz and 32G RAM.

Datasets. We use two real datasets, the road network datasets of Beijing and New York City from the 9th DIMACS Implementation Challenge1 . Each dataset contains an undirected weighted graph that represents a part of the road network. The weight of each edge in a graph represents the distance between two endpoints of the edge. We obtain the keywords of vertices from the OpenStreetMap2 . As shown in Table 2, for D1 in Beijing, we have 168,535 vertices and 196,307 edges. We also have 88,910

1.http://www.dis.uniroma1.it/challenge9/download.shtml
2.https://www.openstreetmap.org

distinct keywords contained by vertices with the total occurrence 1,445,824. For D2 in New York, we have more vertices and edges than D1 in road network with almost twice the size of D1, and the number of keywords contained is larger than D1 as well.

TABLE 2
Statistics of dataset

	Beijing	New York
$\#\|V\|$	168,535	264,346
$\#\|E\|$	196,307	733,846
$\#\|W\|$	88,910	102,450
$\#\|\Phi(V)\|$	1,445,824	3,086,166

Algorithms. We evaluate the performance of three algorithms, greedy clue search algorithm (GCS), clue-based dynamic programming algorithm (CDP) and branch-and-bound algorithm (BAB). In CDP, we use two different distance oracles DO to compute network distance, i.e., all-pair and 2-hop label. In BAB, we evaluate the performances of three index structures, i.e., ABtree, PB-tree and PF.

TABLE 3
Parameter settings

Parameters	Values
Dataset cardinality	4K, 8K, 12K, **16K**
The number of clues	2, 3, **4**, 5, 6, 7, 8
h bits hash code	32, **64**, 128, 256, 512
Average Keyword frequency	10, 50, 100, **500**, 1000, 5000, 10000
Average distance (km)	2, **4**, 6, 8, 10, 12, 14, 16, 18, 20
Average confidence factor	**0.2**, 0.4, 0.6, 0.8, 1.0

TABLE 4
Statistics of keyword frequency on Beijing Dataset

Frequency ≥	10	50	100	500	1000	5000	10000
Percentage (%)	9.72	4.55	3.35	2.25	1.11	0.82	0.61

Parameter settings. We randomly generate 100 queries for each set of experiment and measure their performance by average. To evaluate the algorithms under various settings, we vary the value of some parameters in the query to study the performance, as shown in Table 3. For default settings, we choose 16K for dataset cardinality (the number of vertices), 4 for the number of clues in query, and 64 for hash code length.

We assume a keyword at most shows up once in a vertex, thus the frequency of a keyword w is the number of vertices that contain w, i.e., |Vw|. The statistics of keyword frequency are shown in Table 4, which demonstrates the percentages of keywords with different frequencies. In the query, the keyword frequencies, the average distances and the confidence factors in clues are randomly generated.

More specifically, assume the average keyword frequency for evaluation is |Vw|, thus the keyword frequencies we chose in a clue are in the range [0.9 ∗ |Vw|, 1.1 ∗ |Vw|], which is the similar case with average distance and average.

7.2 Performance Evaluation

Table 5 shows the performance comparison of proposed algorithms and index structures on query time, index size and index construction time. The construction time of all-pair and 2-hop label, which have been studies by existing works, are excluded

TABLE 5
Performance of proposed algorithms and index structures on QT (Query time), IS (Index size) and IT (Index time)

Algorithm		QT (ms)		IS (GB)		IT (min)	
		BJ	NY	BJ	NY	BJ	NY
GCS		14.02	32.17	-	-	-	-
CDP	Allpair	223.81	223.75	106.1	260.6	-	-
	Label	612.72	693.42	0.51	0.78	-	-
BAB	AB-tree	120.84	153.92	856	2104	1045	2569
	PB-tree	**76.12**	**93.27**	**2.1**	**3.21**	**2**	**3.1**
	PF	89.86	106.14	2.2	3.36	2	3.4

in our performance comparison. For the query time evaluation, it is easy to see that BAB well outperforms GCS and CDP. Besides, applying all-pair in CDP has a shorter response time but a larger space cost than utilizing 2-hop label, and using PB-tree in BAB has a better performance than using AB-tree and PF. For index size and construction time, label based approaches have a much smaller size and less time than all-pair based approaches. As NY has a larger size than BJ, more time and space costs are required. For the rest experiments, we only demonstrate the performance on BJ due to the space limit, where the performance on NY is similar to that on BJ.

7.2.1 Accuracy Measurement of GCS

Figure 5 shows the accuracy measurement of GCS by varying the parameters in the query, such as the number of clues, average keyword frequencies, expected distances and . We study the accuracy by two criteria: the matching ratio Amatch and hitting ratio Ahit. Amatch = dm(C, FPgcs) optimal matching distance dmopt Amatch is the ratio of estimated matching distance of GCS on the optimal matching distance. A smaller Amatch means a better accuracy.

Ahit = |FPgcs ∩ FPopt| |C| Ahit focuses on the percentage of match vertices in FPgcs contained by the optimal feasible path FPopt. A greater Ahit means a better accuracy. As we can see in Figure 5, when we enlarge the parameters, Amatch increases as the result of GCS becomes more inaccurate; and Ahit decreases, as less match vertices in optimal feasible path are hit. Both these two criteria becomes less sensitive when the keyword frequency gets larger than 500 in Figure 5(b). Moreover, Ahit keeps steady in Figure 5(c) and 5(d) since GCS is not sensitive to average distance and .

7.2.2 Query Efficiency Measurement

Effect of the keyword hash code length h. In this set of experiments, we study the effect of keyword hash code length h on performance of AB-tree, PB-tree and PF index structures. As shown in Figure 7, the pivot-based indices well outperform AB-tree on index construction time, index size and query time. The space of AB-tree is O(|V | 2 · h) and PB-tree is O(|L| · h). When we enlarge h, both the index size and construction time linearly increase. For query time, there are more false positives in tree traversal when h is small, however, the bit operation costs less time than larger h, which is the case when h is set to 64 comparing with higher values. When we set h to 32, even though we have less bit operation costs, the number of false positives increases such that the query time increases.

Effect of the dataset cardinality. Let us take Beijing dataset for example, we randomly extract 5 subgraphs with equal number of vertices from the original Beijing road network where the performance is measured by average for each experiment. For each subgraph, the connectivity and keyword information of each vertex are kept the same as the original road network. In this set of experiments, we vary the size of these subgraphs to study the performance of proposed algorithms and index structures, as shown in Figure 8. Obviously, the index size and construction time increase when we enlarge the size of datasets. It is worth to note that the size of AB-tree increases exponentially with the number of vertices, and the sizes of PB-tree and PF increase gently especially when the size is enlarged from 120K to 160K due to the property of 2-hop label. For the query time, the BAB algorithm outperforms CDP by a large margin.

Effect of the number of clues. In this set of experiments, Figure 6(a) shows the performance of algorithms by increasing the number of clues in CRS query. Not surprisingly, the response time increases when we enlarge the number of clues of all proposed algorithms. For GCS, the response time increases gently since only more rounds of network expansion are induced. For CDP, when we enlarge the number of clues, more iterations are triggered for the computation. For BAB, the number of candidate vertices and feasible paths increase thus takes more computation time.

Effect of the average frequency of keywords. In this set of experiments, we study the performance of algorithms by varying the frequency of query keywords, as shown in Figure 6(b). It suffices to say that for low frequency keywords, say the frequency less than 500, it is more efficient if we adopt CDP with all-pair, and for high frequency keywords, BAB with PB-tree has a much better performance on both response time and index size. This is because, for CDP, there are not too many combinations to consider if the frequency is low, but when we enlarge the frequency, the response time increases exponentially to the frequency. For BAB, there are lots of false positives if the frequency is low, and when we enlarge the frequency, the

performance becomes much better since we can quickly locate the candidate by using PB-tree.

Effect of the average expected distance. In this set of experiments, we study the effect of average expected distance on the performance of proposed algorithms, as shown in Figure 6(c). As we know, we apply the network expansion algorithm in GCS, which makes it sensitive to the expected distance. When the distance increases, more vertices are involved that results in more computation cost. For CDP with all-pair or label index, they both have a small dependency on the query distance. Therefore, the computation time of CDP keeps almost steady as the distance increases. For BAB, the effect is still not obvious but if the distance is small, we are supposed to find the next candidate more quickly since there are only a small portion of vertices after filtered by distance.

Effect of the average . In this set of experiments, we study the effect of average on the performance of proposed algorithms, as shown in Figure 6(d). When we enlarge the average , more match vertices are considered as candidates, thus the time costs of CDP and BAB increase. For GCS, we can do less network traversal to find the current best match vertex, so the query time reduces when we enlarge .

Evaluation of index updating. Here we evaluate the cost of index

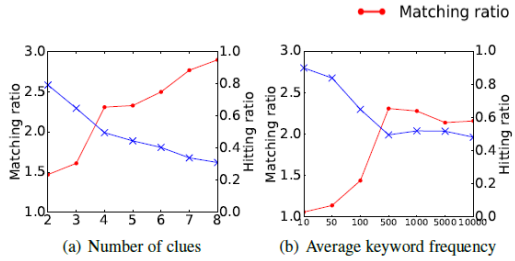


Fig. 5. Accuracy of GCS

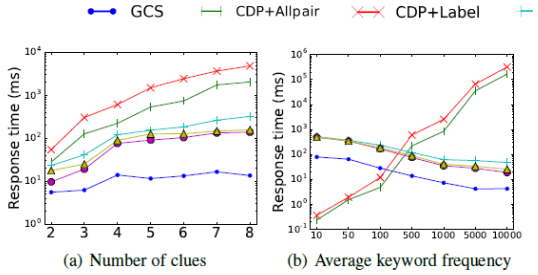


Fig. 6. Query time

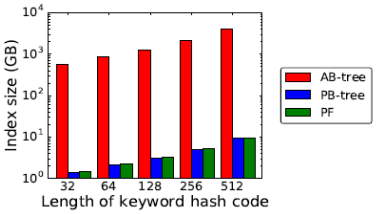
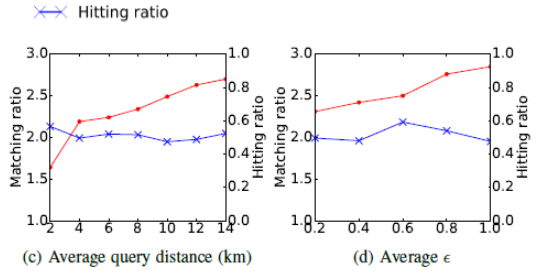


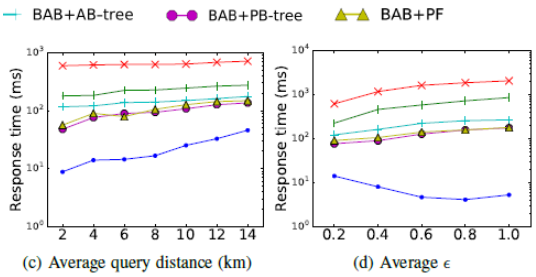
Fig. 7. Effect of the keyword hash code length h

TABLE 6
Evaluation of index updating

Dataset	Update time	Updated pivots
Beijing	78 ms	3.6
NY	127 ms	5.7

keyword hash code 101 102 103 Query time (ms) AB-tree PB-tree PF 32 64 128 256 512 Length of keyword hash code 100 101 102 103 104 Index size (GB) AB-tree PB-tree PF Fig. 7. Effect of the keyword hash code length h TABLE 6 Evaluation of index updating Dataset Update time Updated pivots Beijing 78 ms 3.6 NY 127 ms 5.7 updating. It is easy to observe that the average update time cost is much smaller than reconstruction the index from scratch. The cost comes from two parts, the updating of label index and updating of PF. For each update, we only have to update a very small number of pivot forest structures, that is, the semi-dynamic update is done





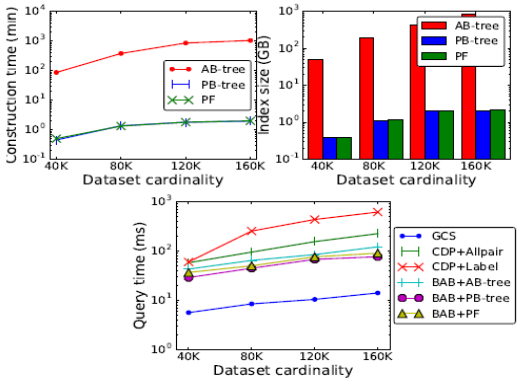


Fig. 8. Effect of the dataset cardinality

VIII. RELATED WORK

In this section, we introduce two lines of related work, top-k spatial keyword search and travel route search.

8.1 Top-k Spatial Keyword Search

Searching geo-textual objects with query location and keywords has gained increasing attention recently due to the popularity of location-based services. In Euclidean space, queries. IR-tree [12] is an R-tree augmented with inverted files that supports the ranking of objects based on a score function of spatial distance and text relevancy. Cao et al. [7] proposes a location-aware top-k prestige-based text retrieval (LkPT) query, to retrieve the top-k spatial web objects ranked according to both prestige-based text relevance (PR) and location proximity. [10] provides an all-round survey of 12 state-of-art geo-textual indices and proposes a benchmark that enables the comparison of the spatial keyword query performance. Zhang et al. [31], [32] proposes the m closet keyword query (mCK query) which aims to find the closest objects that match the query keywords and their distance diameter is minimized. Recently, Guo et al. [15] propose approximation algorithms to solve the mCK query with a ratio of (√ 2 3 +). Cao et al. [8] propose a collective spatial keyword query, in which a different semantics is taken such that the group of objects in the result covers the query keywords and has the lowest cost. Li et al. [23] studies the problem of directionaware spatial keyword search, which aims at finding the k nearest neighbours to the query that contain all input keywords and satisfy the direction constraint. Rocha et al. [27] address the problem of processing top-k spatial keyword queries on road networks where the distance between the query location and the spatial object is the length of shortest path. ROAD [21] organizes the road network as a hierarchy of subgraphs, and connects them by adding shortcuts. For each subgraph, an object abstract is generated for keyword checking. By using network expansion, the subgraphs without intended object are pruned out. G-tree [36] adopts a graph partitioning approach to form a hierarchy. Within each subgraph, a distance matrix is kept, and for any two subgraphs, the distances between all borders of them are stored as well. Based on these distances, it efficiently computes the distance between query vertex and target vertices or tree nodes. Jiang et al. [17] adopt 2-hop label for handling the distance query for kNN problem on large networks, and facilitates KT index to handle the performance issue of frequent keywords. Liu et al. [25] formalize the spatio-textual context information of the querying POI as clues and use them to identify the most promising PoIs, which is closely related to our CRS problem. Different with their work, we aim to find a feasible route on road networks by using clues. In addition, the spatial distance considered in our work is network distance so that the algorithms in [25] can not be applied.

8.2 Travel Route Search

The travel route search problem has been substantially studied for decades. Traveling Salesman Problem (TSP) [11] is the most classic problem in route planning. TSP aims to find the round trip that has the minimum cost from a source point to a set of targets. Li et al. [22] study the problem of Trip Planning Query (TPQ) in spatial databases, where each object is associated with a location and a category. With a starting point S, a destination E and a set of categories C, TPQ retrieves the best trip that starts at S passes through at least one point from each category, and ends at E. TPQ can be considered as a generalization of Travelling Salesman Problem (TSP), thus two approximation algorithms are proposed. [28] studies the problem of optimal sequenced route (OSR), which aims to find a route of minimum length starting from a source point and passing through a number of typed locations in a specific sequence imposed on the types of the locations. They propose a LORD and R-LORD algorithms to filter out the locations that cannot be in the optimal route, thus improves the search efficiency. [9] studies the problem of multi-rule partial sequence route (MRPSR), which aims to find an optimal route with minimum distance under some partial category order rules defined in the query. They propose three heuristic algorithms to search for near-optimal solutions for the MRPSR query. [20] proposes a greedy algorithm to find a route whose length is smaller than a specified threshold while the total text relevance of this route is maximized. [19] studies the problem of finding a route that visits at least one satisfying entity of each type in an interactive approach. In each step, a candidate is given to user to provide a feedback specifying whether the entity satisfies her. [29] studies the problem of multi-approximate-keyword routing query, which complements the standard shortest path search with multiple keywords and an approximate string similarity function. For each keyword, the matching point is supposed to have an edit distance smaller than a given threshold. [6] defines the problem of keyword-aware optimal route query, which is to find an optimal route such

that it covers a set of user-specified keywords, a specific budget constraint is satisfied, and the objective score of the route is optimized. [24] Proposes two different solutions, namely backward search and forward search, to deal with the general optimal route query without a total order. [30] Proposes the problem of personalized trip recommendation, which aims to find the optimal trip that maximizes users' experiences for a given time budget constraint and also takes the uncertain traveling time into consideration.

IX. CONCLUSION AND FUTURE DIRECTIONS

In this paper, we study the problem of CRS on road networks, which aims to find an optimal route such that it covers a set of query keywords in a given specific order, and the matching distance is minimized. To answer the CRS query, we first propose a greedy clue-based algorithm GCS with no index where the network expansion approach is adopted to greedily select the current best candidates to construct feasible paths. Then, we devise an exact algorithm, namely clue-based dynamic programming CDP, to answer the query that enumerates all feasible paths and finally returns the optimal result. To further reduce the computational overhead, we propose a branch-and-bound algorithm BAB by applying filter-and-refine paradigm such that only a small portion of vertices are visited, thus improves the search efficiency. In order to quickly locate the candidate vertices, we develop AB-tree and PB-tree structures to speed up the tree traversal, as well as a semi dynamic index updating mechanism. Results of empirical studies show that all the proposed algorithms are capable of answering CRS query efficiently, while the BAB algorithm runs much faster, and the index size of PB-tree is much smaller than AB-tree. Several directions for future research are promising. First, users may prefer a more generic preference model, which combines PoI rating, PoI average menu price, etc, in the query clue. Second, it is of interest to take temporal information into account and further extend the CRS query. Each PoI is assigned with a opening hours time interval [To, Tc], and each clue contains a visiting time t, where the resulting query aims to find a path such that the time interval of each matched PoI covers the visiting time. Third, requiring users to provide exact keyword match is difficult sometimes as they are just providing "clue", which may be inaccurate in nature. Thus, it is of interest to extend our model to support the approximate keyword match. Hence, the matching distance can be modified by incorporating both spatial distance and textual distance together through a linear combination.

REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In ESA, pages 24–35. Springer, 2012.

[2] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortestpath distance queries on road networks by pruned highway labeling. In ALENEX, pages 147–154. SIAM, 2014.

[3] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In SIGMOD, pages 349–360. ACM, 2013.

[4] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortestpath distance queries on large evolving networks by pruned landmark labeling. In WWW, pages 237–248. ACM, 2014.

[5] J. L. Bentley and J. B. Saxe. Decomposable searching problems i. staticto-dynamic transformation. Journal of Algorithms, 1(4):301–358, 1980.

[6] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. PVLDB, 5(11):1136–1147, 2012.

[7] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. PVLDB, 2010.

[8] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In SIGMOD, pages 373–384. ACM, 2011.

[9] H. Chen, W.-S. Ku, M.-T. Sun, and R. Zimmermann. The multi-rule partial sequenced route query. In SIGSPATIAL, page 10. ACM, 2008.

[10] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. PVLDB, 2013.

[11] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.

[12] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. PVLDB, 2009.

[13] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In ICDE, 2008. [14] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische mathematik, 1(1):269–271, 1959.

[15] T. Guo, X. Cao, and G. Cong. Efficient algorithms for answering the m-closest keywords query. In SIGMOD, pages 405–418. ACM, 2015. [16] C. S. Jensen, J. Kola´rvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In GIS, pages 1–8. ACM, 2003.

[17] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong. Exact top-k nearest keyword search in large networks. In SIGMOD, pages 393–404. ACM, 2015. [18] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. PVLDB, 7(12):1203–1214, 2014.

[19] Y. Kanza, R. Levin, E. Safra, and Y. Sagiv. Interactive route search in the presence of order constraints. PVLDB, 3(1-2):117–128, 2010.

[20] Y. Kanza, E. Safra, Y. Sagiv, and Y. Doytsher. Heuristic algorithms for route-search queries over geographical data. In SIGSPATIAL, page 11. ACM, 2008.

[21] K. C. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In EDBT, 2009.

[22] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In SSTD, pages 273–290. Springer, 2005.

[23] G. Li, J. Feng, and J. Xu. Desks: Direction-aware spatial keyword search. In ICDE, 2012.

[24] J. Li, Y. D. Yang, and N. Mamoulis. Optimal route queries with arbitrary order constraints. TKDE, 25(5):1097–1110, 2013.

[25] J. Liu, K. Deng, H. Sun, Y. Ge, X. Zhou, and C. S. Jensen. Clue-based spatio-textual query. PVLDB, 10(5), 2017.

[26] C. Long, R. C.-W. Wong, K. Wang, and A. W.-C. Fu. Collective spatial keyword queries: a distance owner-driven approach. In SIGMOD, pages 689–700. ACM, 2013.

[27] J. B. Rocha-Junior and K. Nørvag. Top-k spatial keyword queries on ˚ road networks. In EDBT, pages 168–179. ACM, 2012.

[28] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. VLDBJ, 17(4):765–787, 2008.

[29] B. Yao, M. Tang, and F. Li. Multi-approximate-keyword routing in gis data. In SIGSPATIAL, pages 201–210. ACM, 2011.
[30] C. Zhang, H. Liang, K. Wang, and J. Sun. Personalized trip recommendation with poi availability and uncertain traveling time. In CIKM, pages 911–920. ACM, 2015.
[31] D. Zhang, Y. M. Chee, A. Mondal, A. K. Tung, and M. Kitsuregawa. Keyword search in spatial databases: Towards searching by document. In ICDE, pages 688–699. IEEE, 2009.
[32] D. Zhang, B. C. Ooi, and A. K. Tung. Locating mapped resources in web 2.0. In ICDE, pages 521–532. IEEE, 2010.
[33] B. Zheng, N. J. Yuan, K. Zheng, X. Xie, S. Sadiq, and X. Zhou. Approximate keyword search in semantic trajectory database. In ICDE, pages 975–986. IEEE, 2015.
[34] B. Zheng, K. Zheng, X. Xiao, H. Su, H. Yin, X. Zhou, and G. Li. Keyword-aware continuous knn query on road networks. In ICDE, pages 871–882. IEEE, 2016.
[35] K. Zheng, S. Shang, N. J. Yuan, and Y. Yang. Towards efficient search for activity trajectories. In ICDE, pages 230–241. IEEE, 2013.
[36] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In CIKM, 2013.

Authors Profile

Shaik Sharmila pursing M.Tech 2nd year in Qis College and Engineering and Technology in Department of Computer Science and Technology, Ongole.

U. Srinivas Mohan is currently working as an Associate Professor in Department of Computer and Science and Engineering in QIS college of Engineering & Technology with the Qualification M.Tech.